易知收录 原创

硬件遮挡查询

李籽良 发布于 大话事业部

发布时间:2019.02.21 10:27 1239 17 2 更多
分享至POPO眼界大开
已推荐到：职业精选-程序/游戏客户端、今日看点-2.28、今日看点-1.24

遮挡剔除是一种重要的性能优化方法，本文主要分析UE4的GPU硬件遮挡执行流程，方便大家在使用时能更好地理解背后的原理

1 硬件遮挡查询

硬件遮挡查询（Hardware Occlusion Query）指使用GPU来查询物体是否被遮挡，一般会使用一个代理模型或者AABB做查询，如果完全被遮挡，则不渲染物体。流程如下：

1. 创建一个Query对象
2. 关闭RT和Depth的写入
3. 开始查询（Issue Query）
4. 渲染代理模型或者AABB
5. 结束查询
6. 获取查询结果
7. 如果可见像素数目大于0，则渲染物体，否则不渲染

D3D示例代码如下：

```
ID3D11Query *pQuery = nullptr;
D3D11_QUERY_DESC QueryDesc;
QueryDesc.Query = D3D11_QUERY_OCCLUSION;
QueryDesc.MiscFlags = 0;

// 创建Query
g_Device->CreateQuery(&QueryDesc, &pQuery);

// 开始查询
g_DeviceContext->Begin(pQuery);

// 渲染Occlusion模型
RenderOcclusion(m_Model);

// 结束查询
g_DeviceContext->End(pQuery);

bool visible = true;
UINT64 queryData;
// 获取查询结果
while ( S_OK != g_DeviceContext->GetData(pQuery, &queryData, sizeof(UINT64), 0 ) )
{
    // 物体被遮挡
    if(queryData == 0)
        visible = false;
}

if(visible)
{
    RenderModel(m_Model);
}
```

直接使用上述方法会有两个问题：1. 查询时需要场景深度，这意味着在实际渲染之前就要有场景深度。2. 获取查询时CPU需要等待GPU返回，会有较大的延迟。

优化思路是延迟一到两帧做查询，即当前帧的查询结果是之前的，这样可以避免获取深度和延迟的问题，但缺点是高速移动时会有Poping，在实际实现的时候还是有很多细节。下面来看UE4的实现，这里只分析Hardware Occlusion Culling部分，不看Shadow/Reflection的Query，只看正常的BasePass渲染。

2 UE4的实现

下面以简化版的FMobileSceneRenderer::Render源码来看渲染时的Query是如何执行的，我们假设查询延迟为两帧

```
void FMobileSceneRenderer::Render(FRHICommandListImmediate& RHICmdList)
{
    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_SceneStart));

    // 等待第N-2帧的Query
    WaitOcclusionTests(RHICmdList);
    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

    // Find the visible primitives.
    InitViews(RHICmdList);

    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_BasePass));

    // Render base pass
    RenderMobileBasePass(RHICmdList, ViewList);

    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Occlusion));
    // Issue 当前帧的 occlusion queries，N+2帧获取结果
    RenderOcclusion(RHICmdList);

    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
}
```

WaitOcclusionTests会等待当前帧渲染所需的Query，如果RHI是在单独的线程中，那么就需要等待RHI线程已经Issue了当前帧所需的Query（即需要Wait第N-2帧的Issue Query），FSceneRenderer有一个OcclusionSubmittedFence数组，用来存储前N帧的Qurey issue fence，对于当前帧，会Wait第N-2帧的Fence，函数实现如下：

```
void FSceneRenderer::WaitOcclusionTests(FRHICommandListImmediate& RHICmdList)
{
    if (IsRunningRHIInSeparateThread())
    {
        SCOPE_CYCLE_COUNTER(STAT_OcclusionSubmittedFence_Wait);
        // Buffer最后一个，即当前帧所需的Query
        int32 BlockFrame = FOcclusionQueryHelpers::GetNumBufferedFrames(FeatureLevel) - 1;
        // Wait on fence
        FRHICommandListExecutor::WaitOnRHIThreadFence(OcclusionSubmittedFence[BlockFrame]);
        // Query已经Issue，继续执行
        OcclusionSubmittedFence[BlockFrame] = nullptr;
    }
}
```

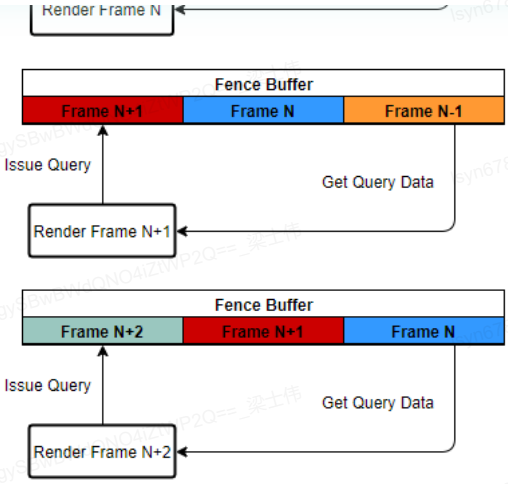
Wait完之后，会在**InitViews**中获取查询结果，然后就能确定每个Primitive的可见性。BasePass完成后，就会调用**RenderOcclusion**，Issue当前帧的Query，RenderOcclusion实现如下：

```
void FMobileSceneRenderer::RenderOcclusion(FRHICommandListImmediate& RHICmdList)
{
    if (!DoOcclusionQueries(FeatureLevel))
    {
        return;
    }

    BeginOcclusionTests(RHICmdList, true);
    FenceOcclusionTests(RHICmdList);

    // Optionally hint submission later to avoid render pass churn but delay query results
    const bool bSubmissionAfterTranslucency = (CVarMobileMoveSubmissionHintAfterTranslucency.GetValueOnRend
    if (!bSubmissionAfterTranslucency)
    {
        RHICmdList.SubmitCommandsHint();
    }
}
```

BeginOcclusionTests为实际的Issue Query逻辑，**FenceOcclusionTests对应WaitOcclusionTests**，如果RHI运行在单独的线程中，那么Issue之后需要创建同步点，然后在N+2帧中Wait这个Fence，函数实现如下：

```
void FSceneRenderer::FenceOcclusionTests(FRHICommandListImmediate& RHICmdList)
{
    if (IsRunningRHIInSeparateThread())
    {
        SCOPE_CYCLE_COUNTER(STAT_OcclusionSubmittedFence_Dispatch);
        int32 NumFrames = FOcclusionQueryHelpers::GetNumBufferedFrames(FeatureLevel);
        // 整个Buffer右移一格，插入当前帧的Fence
        for (int32 Dest = NumFrames - 1; Dest >= 1; Dest--)
        {
            OcclusionSubmittedFence[Dest] = OcclusionSubmittedFence[Dest - 1];
        }
        OcclusionSubmittedFence[0] = RHICmdList.RHIThreadFence();
        RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
    }
}
```

大致流程如下：



3 BeginOcclusionTests

RenderOcclusion这个函数最后调用到FSceneRenderer::BeginOcclusionTests，这里就执行了实际的Issue Query。BeginOcclusionTests函数比较长，这里做了简化，具体细节参考源码。

```
void FSceneRenderer::BeginOcclusionTests(FRHICommandListImmediate& RHICmdList, bool bRenderQueries)
{
    if (bRenderQueries)
    {
        int32 const NumBufferedFrames = FOcclusionQueryHelpers::GetNumBufferedFrames(FeatureLevel);

        // Perform occlusion queries for each view
        for (int32 ViewIndex = 0; ViewIndex < Views.Num(); ViewIndex++)
        {
            // Clear primitives which haven't been visible recently out of the occlusion history, and reset
            ViewState->TrimOcclusionHistory();
        }

        FGraphicsPipelineStateInitializer GraphicsPSOInit;
        RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);
        GraphicsPSOInit.PrimitiveType = PT_TriangleList;
        GraphicsPSOInit.BlendState = TStaticBlendState<CW_NONE>::GetRHI();
        // Depth tests, 关闭RT和深度写入
       GraphicsPSOInit.DepthStencilState = TStaticDepthStencilState<false, CF_DepthNearOrEqual>::GetRHI();
        GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI = GetVertexDeclarationFVector3();

        for (int32 ViewIndex = 0; ViewIndex < Views.Num(); ViewIndex++)
        {
            SCOPED_DRAW_EVENTF(RHICmdList, ViewOcclusionTests, TEXT("ViewOcclusionTests %d"), ViewIndex);

            FViewInfo& View = Views[ViewIndex];
            FViewOcclusionQueries& ViewQuery = ViewQueries[ViewIndex];
            FSceneViewState* ViewState = (FSceneViewState*)View.State;
            SCOPED_GPU_MASK(RHICmdList, View.GPUMask);

           // We only need to render the front-faces of the culling geometry (this halves the amount of pi
           GraphicsPSOInit.RasterizerState = View.bReverseCulling ? TStaticRasterizerState<FM_Solid, CM_CC
            // 是否使用down sample depth
            if (bUseDownsampledDepth)
            {
                // Setup the viewport for rendering to the downsampled depth buffer
                RHICmdList.SetViewport(DownsampledX, DownsampledY, 0.0f, DownsampledX + DownsampledSizeX, D
            }
            else
            {
                RHICmdList.SetViewport(View.ViewRect.Min.X, View.ViewRect.Min.Y, 0.0f, View.ViewRect.Max.X,
            }

            // Lookup the vertex shader.
            TShaderMapRef<FOcclusionQueryVS> VertexShader(View.ShaderMap);
            GraphicsPSOInit.BoundShaderState.VertexShaderRHI = GETSAFERHISHADER_VERTEX(*VertexShader);

            SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);

            if (!ViewState->HasViewParent() && !ViewState->bIsFrozen)
            {
                VertexShader->SetParameters(RHICmdList, View);

                // Issue Query
                {
                    SCOPED_DRAW_EVENT(RHICmdList, GroupedQueries);
                    View.GroupedOcclusionQueries.Flush(RHICmdList);
                }
                {
                    SCOPED_DRAW_EVENT(RHICmdList, IndividualQueries);
                    View.IndividualOcclusionQueries.Flush(RHICmdList);
                }
            }

            // On mobile occlusion queries are done in base pass
```

```
            {
                // Restore default render target
                SceneContext.BeginRenderingSceneColor(RHICmdList, ESimpleRenderTargetMode::EUninitializedCo
            }
        }
    }
}
```

BeginOcclusionTests执行步骤：

1. 清理无效的Query History
2. 生成Graphics PSO用于Query，关闭RT和Depth写入
3. Issue Query
4. 恢复渲染状态

在Feature Level大于ES3.1时，引擎会用一个Down sample的深度作为输入，这样可以提升Query的效率。最后调用Issue Query的是**FViewInfo.IndividualOcclusionQueries和FViewInfo.GroupedOcclusionQueries的Flush函数**，这两个成员的类型是**FOcclusionQueryBatcher**，这个类就存储了真正的RHI Query句柄，并负责创建用来查询的Vertex buffer，Flush的函数定义如下：

```
void FOcclusionQueryBatcher::Flush(FRHICommandList& RHICmdList)
{
    if(BatchOcclusionQueries.Num())
    {
        FMemMark MemStackMark(FMemStack::Get());

        // Create the indices for MaxBatchedPrimitives boxes.
        FIndexBufferRHIParamRef IndexBufferRHI = GOcclusionQueryIndexBuffer.IndexBufferRHI;

        // Draw the batches.
        for(int32 BatchIndex = 0, NumBatches = BatchOcclusionQueries.Num();BatchIndex < NumBatches;BatchInd
        {
            FOcclusionBatch& Batch = BatchOcclusionQueries[BatchIndex];
            FRenderQueryRHIParamRef BatchOcclusionQuery = Batch.Query;
            FVertexBufferRHIParamRef VertexBufferRHI = Batch.VertexAllocation.VertexBuffer->VertexBufferRHI
            uint32 VertexBufferOffset = Batch.VertexAllocation.VertexOffset;
            const int32 NumPrimitivesThisBatch = (BatchIndex != (NumBatches-1)) ? MaxBatchedPrimitives : Nu

            RHICmdList.BeginRenderQuery(BatchOcclusionQuery);
            RHICmdList.SetStreamSource(0, VertexBufferRHI, VertexBufferOffset);
            RHICmdList.DrawIndexedPrimitive(
                IndexBufferRHI,
                PT_TriangleList,
                /*BaseVertexIndex=*/ 0,
                /*MinIndex=*/ 0,
                /*NumVertices=*/ 8 * NumPrimitivesThisBatch,
                /*StartIndex=*/ 0,
                /*NumPrimitives=*/ 12 * NumPrimitivesThisBatch,
                /*NumInstances=*/ 1
                );
            RHICmdList.EndRenderQuery(BatchOcclusionQuery);
        }
        INC_DWORD_STAT_BY(STAT_OcclusionQueries,BatchOcclusionQueries.Num());

        // Reset the batch state.
        BatchOcclusionQueries.Empty(BatchOcclusionQueries.Num());
        CurrentBatchOcclusionQuery = NULL;
    }
}
```

这里我们可以看出引擎在提交时做了优化：**Query是按Batch提交的，每个Batch有16个Primitive**，FOcclusionQueryBatcher通过**BatchPrimitive**函数来收集Primitive的Bound，然后生成对应的Vertex并打成一个Batch。

4 获取Query结果

在BasePass之前，Renderer会调用InitViews，InitViews就执行了Primitive的可见性计算，包括了Frustum Cull，预计算的Visibility和硬件Occlusion Query。引擎最终调用到SceneVisibility.cpp的FetchVisibilityForPrimitives函数来执行Occlusion Query，实际的执行代码很长，包含很多的细节，这里为了理解做了简化：

```
// 遍历VisibilityMap
for (BitIt : View.PrimitiveVisibilityMap)
{
    auto ViewPrimitiveOcclusionHistory = ViewState->PrimitiveOcclusionHistorySet;
    FPrimitiveComponentId PrimitiveId = Scene->PrimitiveComponentIds[BitIt.GetIndex()];

    // 获取到Occlusion history
    auto* PrimitiveOcclusionHistory = ViewPrimitiveOcclusionHistory.Find(FPrimitiveOcclusionHistoryKey(Prim

    if (!PrimitiveOcclusionHistory)
    {
        // If the primitive doesn't have an occlusion history yet, create it.
        PrimitiveOcclusionHistory = &ViewPrimitiveOcclusionHistory[
            ViewPrimitiveOcclusionHistory.Add(FPrimitiveOcclusionHistory(PrimitiveId, SubQuery))
        ];
    }
    else
    {
        uint64 NumSamples = 0;
        bool bGrouped = false;
        FRenderQueryRHIParamRef PastQuery = PrimitiveOcclusionHistory->GetQueryForReading(OcclusionFrameCou
        if (PastQuery)
```

```
                // The primitive is occluded if none of its bounding box's pixels were visible in the previ
                bIsOccluded = (NumPixels == 0);

                if (!bIsOccluded)
                {
                    checkSlow(View.OneOverNumPossiblePixels > 0.0f);
                    PrimitiveOcclusionHistory->LastPixelsPercentage = NumPixels * View.OneOverNumPossiblePi
                }
                else
                {
                    PrimitiveOcclusionHistory->LastPixelsPercentage = 0.0f;
                }
            }
        }
    }
    if (PrimitiveOcclusionHistory)
    {
        bool bRunQuery, bGroupedQuery;
        // 此处细节略去
        //...

        if (bRunQuery)
        {
            // 生成当前帧的Occlusion primitive batch
            PrimitiveOcclusionHistory->SetCurrentQuery(OcclusionFrameCounter,
                bGroupedQuery ?
                View.GroupedOcclusionQueries.BatchPrimitive(BoundOrigin, BoundExtent) :
                View.IndividualOcclusionQueries.BatchPrimitive(BoundOrigin, BoundExtent),
                NumBufferedFrames,
                bGroupedQuery,
            );
        }
    }
}
```

总结一下就是：遍历VisibilityMap，获取到之前的Query句柄，然后调用RHI接口获得查询结果（因为是延迟执行的，那么一般情况下这里不会有太大延迟），这个结果就确定了Primitive的可见性，最后再生成当前帧的Primitive occlusion batch，这批Batch会在之后的RenderOcclusion里提交。

引擎在实际执行的时候会判断是否可以并行计算，如果可以，会生成对应的Graph Task然后并行执行，最大并行Task数目为4；否则就直接在Render线程串行执行。提交Query之前，引擎还会根据历史Occlusion信息来推测是否需要Query，对应的细节太多这里不列出来，详细可参考SceneVisibility.cpp源码。Occlusion历史信息对应**FPrimitiveOcclusionHistory**类，每一个要查询的Primitve就对应一个类。

5 总结

实际完整的Query逻辑很复杂，Deferred和Mobile略有一些不同，但是核心流程依然是延迟执行， UE为了提高Query的性能和效果做了很多的细节优化和平台适配，在效果和效率上做了很多折中，合理运用Occlusion Query能提高游戏的性能。有关内容欢迎POPO讨论。

2楼 个人倾向于使用CPU进行OcclusionCulling的版本，因为GPU绘制几何体本身有GPU开销，而且Query会导致CPU等待GPU，（采用各种Trick优化Draw Call了半天，结果在Query的时候CPU等待GPU，前功尽弃啊2333333）（注：predicate允许GPU不等待CPU，但是层次性的结构需要复杂的逻辑（即Early Out），必须在CPU端进行处理）

2019-02-21 11:13 回复 1

张羽乔 @张羽乔
HZB明显有问题，由于层次性的结构存在，所以需要用CPU去Query处理复杂的逻辑，执行EarlyOut的优化（EarlyOut即：一个父节点不通过，字节点全部不通过），GPU的predicate只能处理非常简单的逻辑（不支持If嵌套，而且GPU的SIMT结构，本身在设计上就是为了处理大量数据，而不是大量的复杂逻辑）；Query会导致CPU等待GPU，好不容易用各种Trick去优化DrawCall（个人对Trick始终持否定态度），结果又在Query里面浪费大量CPU时间，何苦呢?

2019-02-26 19:17 回复 0

张羽乔 @张羽乔 游戏引擎不一定对，从学术权威性角度来讲，大学的Paper > GPU硬件公司的Paper > 游戏引擎

2019-02-26 19:15 回复 0

张羽乔 @张羽乔
根本没有必要用umbra，技术都不公开还要收费，Intel有开源版本Masked-Software-Occlusion-Culling，AVX/SSE移植到NEON应该没多大难度吧

2019-02-26 19:11 回复 0

共 4 条回复，查看更多

张羽乔

1楼 现有的Occlusion Culling主要分两类：
1.Intel的基于CPU的Masked Software Occlusion Culling（http://software.intel.com/en-us/articles/masked-software-occlusion-culling）
2.基于GPU的Hierarchical Z-Buffering，它的思路可能来自GPU硬件实现的DepthTest的方式（注：GPU硬件并不是暴力逐像素进行DepthTest的，而是构造了层次性的结构来加速这个过程）（详见 "Real Time Rendering 4th" 19.7.2 Hierarchical Z-Buffering 中关于HZB的讨论）

个人感觉技术性较强的主题还是参考科研机构（比如Intel、NVIDIA、AMD、各种大学等）的文章比较靠谱，游戏引擎（比如UE4）相对来讲是落后的（而且没有文档，代码还可能还涉及到各种"潜规则"，相对来讲代码也不是很容易看）。

2019-02-21 11:06 回复 0

张羽乔 @张羽乔 awesome!

2019-02-21 11:15 回复 0

李籽良 好的，感谢分享!

2019-02-21 11:13 作者回复 回复 0

加载完毕,没有更多了

该文章被以下专题收入

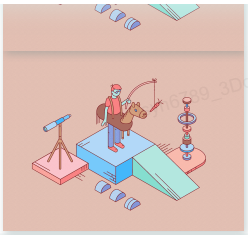
虚幻引擎知识地图（持续更新）
钟钟(钟巧) mkong(孔颖) 宏伟(侯宏伟) 等 编 15670 143个资源

虚幻引擎源码剖析

papalqi(黄琦) 志荣(杜志荣) 编 6733 112个资源

虚幻引擎作为目前世界上最知名且授权最广的顶尖游戏引擎，相比其他引擎，不仅高效、全能，还能直接预览开发效果，赋予了游戏开发者更强的能力。本专题深入虚幻引擎底层实现，通过源码剖析的方式去详细的了解U

最近更新：UE4 SSAO 简述

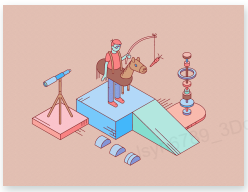


Unreal引擎介绍

罗胜豪 苏汐(苏湘娜) StevenYu(孟玉) 等 编 1845 43个资源

分阶段帮助大家逐步了解虚幻引擎内容，同时收集相关工具支持文档，帮助查阅~

最近更新：（三地）QA进阶培训-Unreal引擎介绍（2021）



大家都在看

《全面战争：战锤》攻城AI系统分析

UE4进阶培训简介

UE5 Nanite 浅析 (一) : 核心思路